CMPT 371 E100

Spring 2025

School of Computing Science, Simon Fraser University

Dr. Shervin Shirmohammadi

April 11, 2025

Project: Report

Group 07

Table of Contents

I. Capture the Flag Game

I.I Objective

In the competitive multiplayer client-server game "Capture the Flag," two teams—Red and Blue—compete to collect strategically placed flags in the environment within a maze-like structure. The grid contains a total of seven flags. Navigating the grid, finding the flags, and attempting to successfully capture them are the goals. Now, to capture the flags that both teams can access at the same time, one must press the "C" button first between 3 to 4 seconds—a time limit that the player must estimate on their own. The person who successfully presses and releases the button within the appropriate time period before any other player captures that specific flag.

I.II Key Features

- Distributed Client-Server Architecture: Implements a robust TCP/IP network architecture ensuring reliable communication.
- Real-Time State Synchronization: Players move using specific keys, and these movements are transmitted to the server and synchronized across all clients to maintain consistency in real-time.
- Shared Object “Flag” Handling: Each flag is a shared resource, meaning multiple players can attempt to capture the same flag simultaneously. The first player to successfully complete a valid capture on a flag—based on timing and server-side confirmation—wins that flag. Once captured, no other player can capture it again.

- Time Capture Mechanic: Capturing a flag requires holding down the ‘C’ key for a random duration between 3 to 4 seconds. This adds an element of skill and timing.
- Dynamic Win-State Evaluation: Continuously monitors game progress to determine victors.

I.III Client-Server Communication

The system employs a centralized server authority model to maintain game state integrity and ensure fair play. This architecture provides several critical advantages:

- Predictable Latency: TCP Protocol guarantees ordered packet delivery.
- Scalable Session Management: The server can concurrently manage multiple game sessions.
- State consistency: All game logic is executed by the server, and all the clients are updated with the current state of the game by the server. Whenever a client changes its state, for example, it moves up a block, it informs the server about its updated state. The server appropriately handles the message from the client and broadcasts the received information to all the clients. In response to that, all the clients update that particular player’s state. This ensures real-time synchronization and consistency.

I.III.I Application Layer Messaging Scheme

- Transport Layer: There are TCP server socket and TCP sockets, each for one client, which are opened when a player starts the game and are pioneers in establishing communication between the server and multiple clients.

- Messaging Protocol: All the communication between server and client occurs in a tokenized manner; each message begins with a keyword token, followed by the actual information. The keyword allows the server or the client to interpret the message and take appropriate steps to update the game's state. For example, a message coming from the server beginning with the keyword "flagCaptured" informed all the clients that a player had captured a flag. A message beginning with "lockFlag" instructs the clients to lock a captured flag and make it unavailable to access.

I.IV. Code Snippets

I.IV.I Opening Sockets

a) The "start ()" method creates server-side socket by opening a server Socket on a specified port and listens for client connections constantly. When a client connects, it accepts the connection, wraps it in Client Handler, adds it to the active client's list, and starts a new thread to check communication with that client. Making this method essential for opening sockets and managing multiple client connections concurrently.

```java
/**
 * starts the server and listens for client connections
 */
public void start() {
    try (ServerSocket serverSocket = new ServerSocket(PORT)) {
        System.out.println("Server listening on port " + PORT);

        while (true) {
            Socket clientSocket = serverSocket.accept();
            System.out.println("New client connected: " + clientSocket.getInetAddress().getHostAddress());

            ClientHandler client = new ClientHandler(clientSocket);
            clients.add(client);
            new Thread(client).start();
        }
    } catch (IOException e) {
        System.err.println("Server error: " + e.getMessage());
    }
}
```

b) The “connectToServer ()” method is an integral part of setting socket processes. It opens a TCP server using the given IP address and port. It helps in establishing input and output streams for better communication and checks for any error during socket connection, which makes this function a crucial and important step in starting a client-server network.

```java
/**
 * Connect to the server
 */
private void connectToServer() throws IOException {
    Socket socket = new Socket(ip, PORT);
    in = new BufferedReader(new InputStreamReader(socket.getInputStream()));
    out = new PrintWriter(socket.getOutputStream(), autoFlush: true);
}
```

c) The “initiate (Stage stage)” method establishes a socket connection to the game server, then asks for player information, and later sets up the GUI. It also starts listening for server messages

and configures the game window with keyboard controls. The method helps in starting a client-side environment over a TCP connection.

```java
/**
 * Initializes the game UI and starts the game.
 *
 * @param stage The primary stage for the game UI
 * @throws IOException if there is an error connecting to the server
 */
public void initiate(Stage stage) throws IOException {

    // Establishes a TCP Connection with the game server
    connectToServer();
    out.println("resendPlayers");

    // Request players' info from the server
    getNumberOfPlayers();
    assert(localPlayer != null);

    // Create UI components
    createUI();
    sendFlagCoordinates();

    // Handles messages coming from the server
    listenForServerMessages();

    // Create scene with keyboard controls
    Scene scene = new Scene(root, width: 800, height: 650);
    setupKeyboardControls(scene);

    // Configure and show the stage
    stage.setTitle("Capture the Flag");
    stage.setScene(scene);
    stage.getIcons().add(new Image(Objects.requireNonNull(Menu.class.getResourceAsStream( name: "/sfu/cmpt371/group7/game/gameIcon.png"))));
    stage.setResizable(false);
    stage.centerOnScreen();
    stage.setOnCloseRequest(e -> {
        Alert alert = new Alert(Alert.AlertType.CONFIRMATION, s: "Are you sure you want to exit?", ButtonType.YES, ButtonType.NO);
        alert.showAndWait().ifPresent(response -> {
            if (response == ButtonType.YES) {
                System.exit( status: 0);
            }
            if (response == ButtonType.NO) {
                e.consume();
            }
        });
    });
    stage.show();
}
```

d) The “broadcast (String message)” method sends a message to all connected clients by iterating through the shared client list. This helps in making sure the code is thread-safe using synchronization and is typically used in socket-based server to update every player in real time.

```java
/**
 * broadcasts a message to all connected clients
 */
private void broadcast(String message) {
    System.out.println("Broadcasting: " + message);

    synchronized (clients) {
        for (ClientHandler client : clients) {
            if (client != null) {
                client.sendMessage(message);
            }
        }
    }
}
```

I.IV.II Handling the Shared Object

a) The “handleCaptureDuration (String [] parts)” method handles the capture logic for a shared object, which is “the flag” in this game setup in a multiplayer setting. Checks if a player successfully captures a flag based on the catching duration and rules of fair gameplay. If it is valid, the method marks the flag captured, updates the team’s scores, notifies all other players or clients and makes the flag unavailable for the others to avoid conflicts. Being part of the handling of shared objects, it helps in maintaining consistent gameplay.

```
/**
 * Handle capture duration message from clients
 * Checks if duration is within valid range (3-4 seconds)
 * If valid, flag is captured and any other players on the flag are respawned
 * If invalid, the attempting player is respawned
 */
private void handleCaptureDuration(String[] parts) {
    // captureDuration <player name> <flag name> <time (sec)>
    if (parts.length >= 4) {
        String playerName = parts[1];
        String flagName = parts[2];
        double duration = Double.parseDouble(parts[3]);

        Flag flagToCapture = findFlagByName(flagName);
        Player attemptingPlayer = findPlayerByName(playerName);

        if (flagToCapture != null && attemptingPlayer != null) {
            // Check if capture duration is within valid range
            double MIN_CAPTURE_DURATION = 3;
            double MAX_CAPTURE_DURATION = 4;
            if (duration >= MIN_CAPTURE_DURATION && duration <= MAX_CAPTURE_DURATION && !flagToCapture.isCaptured()) {
                // Successful capture
                flagToCapture.setCaptured(true);
                broadcast("flagCaptured " + playerName + " " + flagName);

                // Update team score
                if (attemptingPlayer.getTeam().equals("red")) {
                    redFlagCount++;
                } else {
                    blueFlagCount++;
                }
                // Check for other players on the same flag position and respawn them
                for (Player player : PLAYERS) {
                    if (!player.getName().equals(playerName) &&
                            player.getX() == flagToCapture.getX() &&
                            player.getY() == flagToCapture.getY()) {

                        respawnPlayer(player);
                    }
                }
                // Check if this capture results in a win
                checkWinCondition();
            } else {
                // Failed capture - respawn the player
                respawnPlayer(attemptingPlayer);
            }
        }
    }
}   Arora, 2025-04-08, 8:58 p.m. • Implemented the logic of capturing flag after holding C for some time
```

b) The “respawnPlayer (Player player)” method deals with respawning of a player by determining an available spawn location based on the player’s teammates’ location and makes sure no other player occupies the position. It updates the player’s coordinates and broadcasts this change to all connected clients to maintain accuracy in shared state. This method makes sure players re-enter the game properly and fairly.

```java
/**
 * Respawn a player to their team's spawn point
 */
private void respawnPlayer(Player player) {
    int spawnX = 10, spawnY = 10;
    if (player.getTeam().equals("red")) {
        if(isNoPlayerAtPosition( x: 2,  y: 0)) {
            System.out.println("spawning at 0,2");
            spawnX = RED_1_X;
            spawnY = RED_1_Y;
        } else if(isNoPlayerAtPosition( x: 3,  y: 0)){
            System.out.println("spawning at 0,3");
            spawnX = RED_2_X;
            spawnY = RED_2_Y;
        }
    } else {
        if(isNoPlayerAtPosition( x: 2,  y: 19)) {
            System.out.println("spawning at 2,19");
            spawnX = BLUE_1_X;
            spawnY = BLUE_1_Y;
        } else if(isNoPlayerAtPosition( x: 3,  y: 19)){
            System.out.println("spawning at 3,19");
            spawnX = BLUE_2_X;
            spawnY = BLUE_2_Y;
        }
    }

    // Update player position
    player.setX(spawnX);
    player.setY(spawnY);

    // Notify all clients about respawn
    broadcast("respawnPlayer " + player.getName() + " " + spawnX + " " + spawnY);
    broadcast("movePlayer " + player.getName() + " " + spawnX + " " + spawnY);

    System.out.println("Respawning player " + player.getName() + " to " + spawnX + "," + spawnY);
}
```

c) The “listenForServerMessages ()” method continuously listens for incoming messages from the server on a different thread to make real-time updates possible without blocking the main UI. It parses each message by type and starts proper handlers to update clients’ state, for example, starting the game. This method makes client responsiveness perfect through synchronization with the server.

```java
/**
 * Listen for server messages and appropriately handle them
 */
private void listenForServerMessages() {
    new Thread(() -> {
        try {
            String message;
            while ((message = in.readLine()) != null) {
                String[] parts = message.split( regex: " ");
                String messageType = parts[0];

                System.out.println("Received: " + message);

                switch (messageType) {
                    case "movePlayer" -> handleMovePlayerMessage(parts);
                    case "newPlayer" -> handleNewPlayerMessage(parts);
                    case "sizeOfPlayersIs" -> handlePlayerCountMessage(parts);
                    case "gameOver" -> handleGameOverMessage(parts);
                    case "flagCaptured" -> handleFlagCapturedMessage(parts);
                    case "lockFlag" -> handleLockFlagMessage(parts);
                    case "sendingPlayer" -> handlePlayerUpdateMessage(parts);
                    case "playerLeft" -> handlePlayerLeftMessage(parts);
                }
            }
        } catch (IOException e) {
            System.err.println(e.getMessage());
        }
    }).start();
}
```

d) The method “run ()” listens for and handles messages from clients in a loop. Given the type of message, switch cases are activated and messages are handled accordingly, which manages various game scenarios, for example, player movement, team selection, etc. It also handles disconnections and error scenarios, giving robustness for multiplayer interaction.

```java
/**
 * handles incoming messages from the client
 * new and improved switch case statement to handle different message types from the server
 */
@Override
public void run() {
    try {
        String message;
        while ((message = in.readLine()) != null) {
            System.out.println("Received: " + message);
            String[] parts = message.split( regex: " ");
            String messageType = parts.length > 0 ? parts[0] : "";

            switch (messageType) {
                case "teamSelection":
                    handleTeamSelection(parts);
                    sendClientToAllPlayers(message);
                    break;
                case "movePlayer":
                    handleMovePlayer(parts);
                    break;
                case "tellMeTheCurrentPlayers":
                    handleCurrentPlayers();
                    break;
                case "exitGame":
                    handleExitGame(parts);
                    break;
                case "flagCoordinates":
                    handleFlagCoordinates(parts);
                    break;
                case "resendPlayers":
                    handleResendPlayers();
                    break;
                case "gameOver":
                    handleGameOver(parts);
                    break;
                case "captureDuration":
                    handleCaptureDuration(parts);
                    break;
                default:
                    System.out.println("i dont know what you mean. when you wanna say less but you wanna say no" + messageType);
                    break;
            }
        }
    } catch (IOException | InterruptedException e) {
        System.err.println("Error in client handler: " + e.getMessage());
    } finally {
        handleDisconnect();
    }
```

II. Group Members and Contributions

This group project was developed collectively by the following team members: Ayush Arora, Daksh Arora, Kabir Singh Sidhu and Hetmay Ketan Vora.

Below are the details of the individual responsibilities and contributions.

Name	Role	Contribution
Kabir Singh Sidhu	Server Architecture and Network Communication	25%
Daksh Arora	Game GUI and map Rendering	25%
Ayush Arora	Synchronization and Flag Capture Logic	25%
Hetmay Ketan Vora	Testing, Debugging, and Documentation	25%

Key Responsibilities

Kabir Singh Sidhu: Built the server, managed socket connections, and handled message broadcasting.

Daksh Arora: Handled player movement, GUI setup, map logic and rendering.

Ayush Arora: Implemented flag capture logic, team scoring, and game state synchronization.

Hetmay Ketan Vora: Conducted testing, resolved bugs, and prepared the final report.

III. Source Code

Source code along with appropriate comments, can be accessed through the GitHub link given below:

GitHub Repository Link: CMPT371-Group07

IV. Demo Video

A Video demonstration showcasing the game's functionality can be accessed through the YouTube link given below. The video features:

- Player movement and logic
- The shared flag mechanism in play.
- Four players competing to capture flags in real time.
- Team winning after capturing four flags.

Video Link: Game Video Demo